3: Sorting Customer Orders

Understanding Sorting Algorithms

Bubble Sort:

Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

- **Time Complexity**:
    - Best Case: O(n)
    - Average Case: O(n^2)
    - Worst Case: O(n^2)
- **Space Complexity**: O(1) (in-place sort)

Example: Consider the array [5, 2, 9, 1, 5, 6].

- **First Pass**:
    - Compare 5 and 2 -> swap -> [2, 5, 9, 1, 5, 6]
    - Compare 5 and 9 -> no swap -> [2, 5, 9, 1, 5, 6]
    - Compare 9 and 1 -> swap -> [2, 5, 1, 9, 5, 6]
    - Compare 9 and 5 -> swap -> [2, 5, 1, 5, 9, 6]
    - Compare 9 and 6 -> swap -> [2, 5, 1, 5, 6, 9]
- **Second Pass**:
    - Compare 2 and 5 -> no swap -> [2, 5, 1, 5, 6, 9]
    - Compare 5 and 1 -> swap -> [2, 1, 5, 5, 6, 9]
    - Compare 5 and 5 -> no swap -> [2, 1, 5, 5, 6, 9]
    - Compare 5 and 6 -> no swap -> [2, 1, 5, 5, 6, 9]
- **Third Pass**:
    - Compare 2 and 1 -> swap -> [1, 2, 5, 5, 6, 9]
    - Compare 2 and 5 -> no swap -> [1, 2, 5, 5, 6, 9]
    - Compare 5 and 5 -> no swap -> [1, 2, 5, 5, 6, 9]

The array is now sorted.

Insertion Sort:

Insertion Sort builds the final sorted array one item at a time. It takes each element and inserts it into its correct position among the elements that have already been considered.

- **Time Complexity**:
  - Best Case: O(n)
  - Average Case: O(n^2)
  - Worst Case: O(n^2)
- **Space Complexity**: O(1) (in-place sort)

Example: Consider the array [5, 2, 9, 1, 5, 6].

- **First Pass: [5, 2, 9, 1, 5, 6]**
  - **The first element is already sorted.**
- **Second Pass: [5, 2, 9, 1, 5, 6]**
  - **Insert 2 into the sorted array [5].**
  - **Result: [2, 5, 9, 1, 5, 6]**
- **Third Pass: [2, 5, 9, 1, 5, 6]**
  - **Insert 9 into the sorted array [2, 5].**
  - **Result: [2, 5, 9, 1, 5, 6]**
- **Fourth Pass: [2, 5, 9, 1, 5, 6]**
  - **Insert 1 into the sorted array [2, 5, 9].**
  - **Result: [1, 2, 5, 9, 5, 6]**
- **Fifth Pass: [1, 2, 5, 9, 5, 6]**
  - **Insert 5 into the sorted array [1, 2, 5, 9].**
  - **Result: [1, 2, 5, 5, 9, 6]**
- **Sixth Pass: [1, 2, 5, 5, 9, 6]**
  - **Insert 6 into the sorted array [1, 2, 5, 5, 9].**
  - **Result: [1, 2, 5, 5, 6, 9]**

The array is now sorted.

Quick Sort:

Quick Sort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

- **Time Complexity**:
  - Best Case: O(nlogn)
  - Average Case: O(nlogn)
  - Worst Case: O(n^2)
- **Space Complexity**: O(logn) (due to recursive stack calls)

Example: Consider the array [5, 2, 9, 1, 5, 6].

- **First Partition** (pivot=5):
  - Elements less than 5: [2, 1]
  - Elements greater than 5: [9, 6]
  - Result: [2, 1] + [5] + [9, 6]
- **Sort Left Sub-array** ([2, 1]):
  - Pivot=2:
    - Elements less than 2: [1]
    - Result: [1] + [2]
  - Left sorted sub-array: [1, 2]
- **Sort Right Sub-array** ([9, 6]):
  - Pivot=9:
    - Elements less than 9: [6]
    - Result: [6] + [9]
  - Right sorted sub-array: [6, 9]
- **Combine**:
  - Result: [1, 2] + [5] + [6, 9] = [1, 2, 5, 5, 6, 9]

The array is now sorted.

Merge Sort:

Merge Sort is also a divide-and-conquer algorithm. It divides the array into two halves, recursively sorts them, and finally merges the two sorted halves.

- **Time Complexity**:
  - Best Case: O(nlogn)
  - Average Case: O(nlogn)
  - Worst Case: O(nlogn)
- **Space Complexity**: O(n) (not an in-place sort)

Example: Consider the array [5, 2, 9, 1, 5, 6].

- **Divide**:
  - Left half: [5, 2, 9]
  - Right half: [1, 5, 6]
- **Sort Left Half** ([5, 2, 9]):
  - Divide into [5] and [2, 9]
  - Sort [2, 9]:
    - Divide into [2] and [9]
    - Merge to get [2, 9]
  - Merge [5] and [2, 9]:
    - Result: [2, 5, 9]
- **Sort Right Half** ([1, 5, 6]):
  - Divide into [1] and [5, 6]
  - Sort [5, 6]:
    - Divide into [5] and [6]
    - Merge to get [5, 6]
  - Merge [1] and [5, 6]:
    - Result: [1, 5, 6]
- **Merge**:
  - Merge [2, 5, 9] and [1, 5, 6]:
    - Result: [1, 2, 5, 5, 6, 9]

The array is now sorted.

Setup and implementation

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Order;

import java.util.*;

/**
 *
 * @author Aishwarya
 */
public class Order {

    int orderId;
    String Customer_name;
    int totalPrice;

    Order(int orderId, String Customer_name, int totalPrice){
        this.orderId = orderId;
        this.Customer_name = Customer_name;
        this.totalPrice = totalPrice;
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter no of Orders : ");
        int n = sc.nextInt();
        Order[] obj = new Order[n];
        for(int i = 0; i < n; i++){
```

```
        System.out.println("Enter Order id : ");
        int o = sc.nextInt();
        System.out.println("Enter Customer_name : ");
        String name = sc.next();
        System.out.println("Enter Total Price : ");
        int totalPrice = sc.nextInt();
        obj[i] = new Order(o,name,totalPrice);
    }

    System.out.println("Enter your choice for sorting (Bubble sort : 0 / Quick sort : 1)");
    int choice = sc.nextInt();

    if(choice == 1){
    int low = 0;
    int high = obj.length-1;
    quick(obj,low,high);
    display(obj);
    }
    else{
        Bubblesort(obj);
        display(obj);
    }

}

 public static void quick(Order[] arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quick(arr, low, pi - 1);
        quick(arr, pi + 1, high);
    }
}

public static int partition(Order[] arr, int low, int high) {
    int pivot = arr[high].totalPrice;
    int i = (low - 1);

    for (int j = low; j < high; j++) {
        if (arr[j].totalPrice >= pivot) {
            i++;

            int temp = arr[i].totalPrice;
            arr[i].totalPrice = arr[j].totalPrice;
            arr[j].totalPrice = temp;

             String tname = arr[i].Customer_name;
            arr[i].Customer_name = arr[j].Customer_name;
            arr[j].Customer_name = tname;

             int temp1 = arr[i].orderId;
            arr[i].orderId = arr[j].orderId;
            arr[j].orderId = temp1;


        }
    }

    int temp = arr[i + 1].totalPrice;
    arr[i + 1].totalPrice = arr[high].totalPrice;
    arr[high].totalPrice = temp;

     String tname = arr[i+1].Customer_name;
            arr[i+1].Customer_name = arr[high].Customer_name;
            arr[high].Customer_name = tname;

             int temp1 = arr[i+1].orderId;
            arr[i+1].orderId = arr[high].orderId;
            arr[high].orderId = temp1;

    return i + 1;
}



public static void Bubblesort(Order[] arr){

    for(int i = arr.length-1; i > 0; i--){
            for(int j = 0; j < i; j++){
                if(arr[j].totalPrice < arr[j+1].totalPrice){
                    int temp = arr[j].totalPrice;
                    arr[j].totalPrice = arr[j+1].totalPrice;
                    arr[j+1].totalPrice= temp;



                 String tname = arr[i].Customer_name;
                arr[i].Customer_name = arr[j].Customer_name;
                arr[j].Customer_name = tname;

                 int temp1 = arr[i].orderId;
                arr[i].orderId = arr[j].orderId;
                arr[j].orderId = temp1;
                }
            }
        }
    }

    public static void display(Order[] obj){
        int n = obj.length;

        for(int i = 0; i < n; i++){
            System.out.println("Order id : "+ obj[i].orderId);
            System.out.println("Customer name : "+ obj[i].Customer_name);
            System.out.println("Total Price : " + obj[i].totalPrice);
        }
    }

}
```

Analysis: Bubble Sort vs. Quick Sort

Performance Comparison:

- **Time Complexity**:
  - Bubble Sort:
    - Best Case: O(n) (when the list is already sorted)

        - Average/Worst Case: O(n^2)
    - Quick Sort:
        - Best Case: O(nlogn)
        - Average Case: O(nlogn)
        - Worst Case: O(n^2) (when the pivot selection is poor, such as always picking the largest or smallest element as the pivot)

Why Quick Sort is Generally Preferred Over Bubble Sort:

1. Quick Sort has an average-case time complexity of O(nlogn), which makes it much more **efficient** for large datasets compared to Bubble Sort's O(n^2) average-case complexity.
2. In practical implementations, Quick Sort is **faster** than Bubble Sort due to better cache performance and fewer swaps.
3. Quick Sort's **divide-and-conquer** approach is more sophisticated and allows for better optimization opportunities, such as parallel processing.
4. Quick Sort can be easily **adapted** to work with different pivot selection strategies to minimize the worst-case scenario, making it more robust for various input cases.